Secure Synthesis of Code: A Process Improvement Experiment

P. Garbett[1], J.P. Parkes[1], M. Shackleton[1⋆], and S. Anderson[2]

[1] Lucas Aerospace, York Road, Hall Green, Birmingham B28 8LN, UK,
shacklm@liyorkrd.li.co.uk,
Tel: +44 121 627 6600, Fax: +44 121 607 3619
[2] Division of Informatics, University of Edinburgh, Edinburgh EH9 3JZ, UK,
soa@dcs.ed.ac.uk,
Tel: +44 131 650 5191, Fax: +44 131 667 7209

Abstract. Arguments for and against the deployment of formal methods in system design are rarely supported by evidence derived from experiments that compare a particular formal approach with conventional methods [2]. We illustrate an approach to the use of formal methods for secure code synthesis in safety-critical Avionics applications. The technique makes use of code components and uses sound introduction rules for the components to ensure constraints on their use are enforced. The approach we describe is the subject of a controlled experiment where it is running in parallel with the conventional approach. We describe the experiment and report some preliminary findings.

1 Introduction

Lucas Aerospace[1] develop safety-critical avionics software. In particular, we have a long history of constructing Full Authority Digital Engine Controllers (FADECs) that control the fuel supply to aircraft engines. "Full Authority" means that there is no reversionary (backup) control. This means the digital control system is a critical function. In an earlier paper [8] we described Lucas Aerospace's overall approach to process improvement. Process improvement is necessary in highly critical software production because we would like to see a reduction in life-cycle costs for new, more sophisticated designs while maintaining or improving on the safety integrity of the product.

In mature development processes it is necessary to innovate in the process to achieve significant improvements. Because untried techniques carry implementation risk we have developed an approach to process innovation based on

⋆ Partially supported by the EU ESSI programme projects, no. 23743, *Proof by Construct using Formal Methods* and no.27825, *Implementing Design Execution using Ada*.

[1] Lucas Aerospace is a TRW company.

J. Wing, J. Woodcock, J. Davies (Eds.): FM'99, Vol. II, LNCS 1709, pp. 1816–1835, 1999.

experimentation with new techniques. This paper describes our approach to process innovation in one facet of our current process and the ongoing experiment to assess the effectiveness of the approach.

The aspect of our development process we consider here is the generation of code from a specification. We use formal development to synthesise code from the specification. The aim of the experiment is to see the extent to which using formal methods can control errors arising in code production. We believe there will be a measurable difference between our formal approach and the existing approach.

1.1 Safety Critical Avionics Software

The main characteristics of safety-critical avionics software are: A *high reliability requirement*, typically the required system reliability of software components is better than one failure in 10^6 flying hours. The controller and equipment under control are *co-engineered*, this leads to much greater variability of requirements through the lifetime of the system. As aircraft employ more digital systems the *size and complexity* of individual sub-systems tends to grow. Software produced by Lucas Aerospace must *certified* to comply with Level A of DO178B [14]. Developing software that complies to such exacting standards take a great deal of effort — safety critical software is expensive to produce.

In justifying any claim of high-reliability the producer of the system needs to (explicitly or implicitly) make a safety case for the system. The safety case must be: *a demonstrable and valid argument that a system is adequately safe over its entire lifetime* [1]. Any safety case will deploy three different kinds of argument in its support: *Deterministic* rule-based, logical arguments that are used to argue that some state of affairs is certainly true. The current Lucas Aerospace process rests to some extent on such elements[12]. *Probabilistic* arguments based on data drawn from testing, field experience etc. *Qualitative* arguments based on showing good practice or conformance to DO178B, and ISO 9000-3.

Generally a well-balanced safety case will involve diverse arguments of all types. In the case of high-reliability systems diversity is essential. Lucas Aerospace already have some experience in the use of formal verification to move the burden of evidence in the safety case from test to proof. This experiment aims to move the demonstration of the absence of certain classes of coding error from testing to the point at which code is synthesised. We believe this could result in significant cost reductions without compromising the safety and reliability of the products. Earlier discovery of errors in requirements and development rather than waiting until test is often claimed to be one of the principal benefits of formal methods [5, 9]

The seeds of the applicability of formal methods lie in the restrictions inherent in the approach to coding and design embodied in the current development process. In the next section we highlight those aspects of the current process that facilitate the use of formal methods.

1.2 The Current Approach

The current development process used by Lucas Aerospace has been refined for over a decade. It produces highly reliable systems and is stable and well understood. We are therefore keen that any proposed change retains the best features of existing processes whilst gaining the benefits of greater formality.

In the current approach, project staff implement requirements by composition of highly verified microprocessor specific elements [3]. This is codified in the domain specific language LUCOL[2] [12].

LUCOL elements are developed by specialists. The development of these elements is via a process which is mature, monitored by metrics which are used for feedback, and supported by highly developed verification processes, including formal verification [12]. That is, it satisfies many of the properties required to achieve the higher levels of the Capability Maturity Model (CMM) [17].

The LUCOL language embodies many constraints that facilitate informal validation and verification processes:

- Design is carried out at the level of combining LUCOL modules to provide high-level functionality.
- Individual LUCOL modules consist of up to around 100 lines of assembler. These are formally verified using the SPADE tool [3]. Loops can occur within these modules and formal verification guarantees their partial correctness. Confidence of termination derives from design guidelines, informal reasoning and extensive test. These modules are also reused across projects and have considerable operational evidence for correctness and reliability.
- LUCOL modules have determinate space, time and functional characteristics and this is inherited by programs constructed from LUCOL modules.
- LUCOL programs are loop-free. The acyclic structure guarantees termination provided each module terminates. The acyclic structure is iterated by a single control loop containing a cyclic executive that controls lower level functions.
- LUCOL programs are executed cyclically. The program is similar to a large filter that is executed at discrete time intervals (frequently enough to ensure adequate control).
- Coding conventions ensure each variable is defined once and there are strong controls over "non-functional" interactions between modules.
- Diagrammatic output can be generated from the code, and then compared to the requirements, which are very often of a similar form.

Taken together, these constraints make the code very amenable to dynamic test. This is particularly true if no historic data sources are kept. These amount to the introduction of feedback loops in the LUCOL program. The absence of loops means that there is no dependence on sequences of events or values, so the function mapping inputs to outputs is time independent. This feature also eases formal proof. Unfortunately it is impossible to eliminate feedback completely so

[2] LUCOL is a trademark of Lucas Aerospace.

we need good tools to analyse programs with feedback. We believe that formal approaches provide such tools to control and analyse these feature of programs.

These considerations mean that LUCOL programs exhibit many of the characteristics of circuits. This suggested that, in addition to considering formal methods tools that support software development, we should consider formal methods tools for hardware development. The resulting approach is aimed at supporting the development of software whose structure is constrained in a manner similar to LUCOL. The approach is language independent and the work is intended to target either LUCOL or Ada.

2 Formal Code Synthesis

2.1 Choice of Formal Methods Tool

The following considerations shaped our choice of formal methods tool. The final decision to choose LAMBDA[3] [16, 15, 11, 7, 6] was taken because it appears to be the most mature industrial-strength tool that matches most of these requirements.

- The core functionality of controllers is specified as a control law expressing the relationship between input and output over time. *Signals* varying over time are conveniently expressed as functions from time to data values. Thus a language with conveniently expressible functional forms is attractive because it is natural to describe our systems as functionals over a set of input and output signals.
- Because we want to reason about functionals it is desirable to use some form of higher logic or type theory.
- The cyclic nature of the system suggests that in most circumstances a loop invariant is the most natural statement of the functional requirements.
- An assumption of the development process is that a top level decomposition of the software into components would split this large invariant into subpieces which apply to sub-components.
- The actual implementations we have, which have had exposure to millions of hours of in-service life, show that it is possible to build extremely effective systems given highly restricted means of altering the flow of control, in particular loopless construction. These structures are particularly easy to formalise in higher-order logic.
- The majority (all but one) of loops can be placed in reusable components, these can be modelled functionally using recursion. These are the most difficult of the specifications to construct (especially given that they are ideally primitive recursive) so are best left to specialists. Ideally we would like to make the internal structure of components (e.g. LUCOL modules) invisible to designers who use the components to construct programs. The method should support the factoring of proof effort between specialist and the development team.

[3] LAMBDA is a trade mark of Abstract Hardware Limited

– An important aspect of any controller design is the control of when a component is active. This is important both for scheduling activity and for sequencing of control modes of the system. An important requirement was a simple method to specify activation/deactivation of components. This is an essential prerequisite to allow the development of any part of a realistic control system.

LAMBDA using poly/ML and the L2 logic was chosen as a vehicle for implementing the functionally based specifications of the components and programs. Although targeted at hardware techniques the method of specification is very similar to that used for controllers. In choosing LAMBDA we felt that it fitted our needs particularly well.

The LAMBDA hardware synthesis method is predicated on the approach that formal techniques could be made as transparent as possible to the project user. This therefore both technically and otherwise supports the factoring of expertise which is a positive attribute of the existing development process. This division of activity between formal methods experts and developers is reflected in the following two sections. In Section 2.3 we consider the activity of developing a formally verified "library" of code components along with formally defined rules for their use. In Section 2.4 we consider using the predefined rules to structure code synthesis.

In our preliminary investigations of the feasibility of the approach, software involving reactive control components was used as a test case for the method. The original motivation for this work was to capture control diagrams, and this is where it is most effective. The essentially discrete switching logic controlling these functional programs has been targeted by other formal methods, e.g. they could be approached using formalisms based on state machine diagrams.

2.2 An Overview of LAMBDA

This section is intended to provide enough background to LAMBDA to allow the reader to understand the notation used in the technical sections of the paper. We split the account into three sections dealing with the things one can define and manipulate in LAMBDA, how to express goals and theorems in LAMBDA and how to represent the current state of a proof.

LAMBDA Terms The L2 logic used in LAMBDA is a type theory with higher order objects. The language used to define terms in the logic is very similar to the purely functional fragment of the Standard ML programming language [10]. This provides a convenient means to define the behaviour of code components used in the synthesis of controllers. The higher order features of the term language are particularly useful in defining functions that have *signals* as inputs and generate a signal as output. For example the function:

```
fun x ladd y = fn t => wrange(x t ++ y t) ;
```

defines an infix operation `ladd` that takes two signals `x` and `y` as input and has the signal whose value at time `t` is the sum of the values of signals `x` and `y` at time `t`.

LAMBDA Sequents The truth of any proposition in the LAMBDA logic is always relative to a context that defines the terms used in the proposition. The context may have dependencies within it so the order of terms in the context is important. For example the axiom asserting reflexivity is given by:

```
G // P $ H |- P
```

The context to the left of the turnstile is called the *assumptions* and the proposition to the right of the turnstile is called the *assertion*. Often we want to think of sequents as *schematic*. In the reflexivity axiom the `P` stands for an arbitrary proposition and the `G` and `H` stand for the rest of the context to the left and right of the assumption `P`.

LAMBDA Rules LAMBDA represents the state of an incomplete proof using the notion of a derived rule in the LAMBDA logic. This notion was first introduced by Paulson [13] in his proof assistant Isabelle. The mechanism of constructing new schematic rules from the basic set of rules in the logic is fundamental to LAMBDA. A typical rule is the and introduction rule:

```
 G//H |- P
 G//H |-    Q
---------------
 G//H |- P/\Q
```

This has the usual interpretation that if the two *premises* above the line are true then the *conclusion* below the line is true. New derived rules are constructed by unifying the conclusion of some rule with one of the premises of another rule. The process of proving some theorem then proceeds by repeatedly carrying out this procedure until the required rule is derived (provided it is derivable). For more details we refer the reader to the references on LAMBDA.

2.3 Reduction of Code to Component Form

In this section we consider the activity of developing the basic library of components from which programs will be synthesised and establishing sound introduction rules in the LAMBDA logic that admit the use of the components in synthesising code. This work requires detailed knowledge of the LAMBDA logic and is usually carried out by formal methods experts.

The code synthesis technique can target either the LUCOL or Ada language. Since Ada is the more generally known, the discussion will here be restricted to Ada. It is worth mentioning in passing that because Ada is compiled language, even if the Ada source is formally synthesised from its specification there is no guarantee that the machine code emitted by the compiler will in turn conform

to that specification. In contrast, no machine code is generated when targeting to the LUCOL language; all of the code resides within the hand-coded modules which are proved to conform to their pre- and postconditions. The LUCOL approach eliminates the need to depend on a large and complex tool.

```
fun x ladd y = fn t => wrange(x t ++ y t) ;

function add (input1 : short_integer;
              input2 : short_integer) return short_integer
--# pre  true;
--# post add = wrange (input1 + input2);
is
    outval : integer;
    output : short_integer;
begin
    outval := integer (input1) + integer (input2);
    if outval < integer (short_integer'first) then
        output := short_integer'first;
    elsif integer (short_integer'last) < outval then
        output := short_integer'last;
    else
        output := short_integer (outval);
    end if;
    return output;
end add;
```

Fig. 1. Specification of Protected Addition

Whilst functional programming languages possess many attractive qualities, for the foreseeable future imperative languages will continue to be used in embedded systems. Imperative functions without side effects or pre-conditions match the semantics of their functional counterparts fairly closely. Figure 1 shows an example of an L2 specification of a "protected addition" and a counterpart written in Ada. In this context protected means that overflows are mitigated by clamping at the appropriate extreme of the 16 bit two's complement integer range. The L2 specification takes two *signals* as arguments and generates a signal as result. Recall that a signal is a map from time (here modelled by the natural numbers) to data values. Despite the simplicity of the function, it is not immediately obvious that the imperative code and the applicative specification perform the same operation. The formal comments (introduced by `--#`) in the Ada code giving the precondition and postcondition for the `add` function are more useful in this respect than the code itself, provided that they have been formally verified. Having conducted such a proof, one might feel justified in believing that the Ada function `add` implements the L2 "software component" whose specification is

```
val ADD#(input1 : int, input2 : int, output int) =
                        output == wrange(input1 ++ input2);
```

More will be said about this correspondence later when modelling the code in L2 is considered. For the moment the point is that one can replace a sub-expression of the form

```
x ladd y
```

with the output z of the software component[4]

```
ADD#(x t, y t, z t)
```

knowing that there is some code available which corresponds to it.

Our approach is: To construct a detailed low-level specification of the required functions in L2. This specification is type correct and is close in form to the informal requirements for the system. Then replace each subexpression in the L2 specification by a software component which has an imperative implementation. This is carried out under the control of LAMBDA and so each replacement must be justified in the logic. Finally during code generation each software component is replaced by the corresponding implementation to derive the code for the system.

Because the limitations on what can be done efficiently in imperative code, notably the limited integer word length, are captured accurately in LAMBDA's logic the specifier must take account of all boundary conditions and other implementation constraints as they synthesise the component form from the L2 specification.

The replacement of subexpressions by software components is a process of replacing L2 by L2 which is performed formally. Such replacements are achieved by an "introduction rule". An introduction rule for the `ADD#` component might be of the form

```
G // H |- P#(o)
-------------------------------------------------
G // forall t. ADD#(x t, y t, o t) $ H |- P#(x ladd y)
```

This rule says that if we connect up an `ADD with` inputs `x` and `y` and output `o` then any occurrence of a sub-expression of the form `x ladd y` may be replaced by `o`. In this instance we are committed to calling our `ADD` component for all time ie. on every cycle of the program, which is not always what is required. Other forms of introduction rules only require output from a component at times when a given set of conditions holds.

Introduction rules are proved to be sound. This guarantees that a component cannot be used incorrectly. The introduction rule for the `ADD` component is very

[4] Note that we are being a little imprecise here. Strictly we would need to be replacing `(x ladd y)` in a context where it is applied to `t` or the `ADD#` relation would have to hold for all times `t`.

easily proved since the component definition differs little from the L2 function it replaces. In some cases the proof can be quite complex.

The simplicity of the ADD example suggests two things; firstly that this is not getting us very far, and secondly that every sub-expression of the L2 specification is going to end up as a subprogram. To illustrate that the former is not necessarily the case we consider a slightly more interesting function. A major strength of an L2 functional specification is that it is free of the state which exists in an imperative program. Thus a simple rectangular integrator may be specified as

```
fun intgr x gain init 0   = init ++ gain ** x 0
  | intgr x gain init 1't = intgr x gain init t ++ gain ** x 1't;
```

The corresponding software component might be

```
val INTGR#(input:int,gain:int,store:int signal,output:int,t:time)=
        output == store t ++ gain ** input /\ store 1't == output;
```

This is not easily implementable as imperative code, of course, because it requires unbounded integer arithmetic and a practical definition of the L2 function intgr would restrict word lengths, but it does illustrate the introduction of state into the software component. The introduced state here is the integer signal store. Whereas the L2 function is defined recursively, the software component introduces a state variable to eliminate the recursion. The corresponding imperative code would have the (infeasible due to limited length arithmetic) specification

```
procedure intgr(input: in short_integer; gain: in short_integer;
           store: inout short_integer; output: out short_integer)
--# pre true;
--# post output = store~ + gain * input and store = output;
```

In order for the imperative code to be able to implement the requirement of the software component that store 1't == output, each use of the integrator function must have associated with it a separate statically allocated state variable which preserves its value between successive calls to the procedure, recording the value of output on the last iteration. Furthermore, the output of the L2 specification of intgr at time 0 requires that the state variable is initialised at time 0. An introduction rule for this component might be

```
G // H |- P#(o) /\ store 0 == init
------------------------------------------------------------
G // forall t. INTGR#(i t,gain,store,o t,t) $ H |-
                                        P#(intgr x gain init)
```

This rule could not be proved without the extra condition that store is initialised at time 0, and ensures that this initialisation constraint is introduced at the same time as the software component. As development proceeds the formal structure of LAMBDA gathers all such constraints and maintains them so the developer has a clear statement of the constraints under which the code can implement the L2 specification.

When the target language is Ada, software components can be implemented by inline code. For instance, the code fragment

```
--# pre divisor <> 0
output := input / divisor;
--# post output = input div divisor
```

can straightforwardly be used to implement integer division, assuming the parameters have the correct Ada types. Where preconditions are other than true, as here, they appear as a side-condition in the corresponding introduction rule, requiring their discharge as part of the code synthesis process.

So far, only component replacements for all time instants have been considered. For instance, when we compute

```
out1 t = if select t then fn1 in1 t else fn2 in1 t
```

it is certainly possible to compute the value of both `fn1` and `fn2` on all cycles and then to discard the value which is not required according to the value of boolean `select t`. This does not make for very efficient code. To avoid this we need a nested introduction rule. For example, such a rule for the `ADD` component is

```
G // H |- if select1 t then P#(o t) else Q
------------------------------------------------------------
G // forall t. select1 t == true ->> ADD#(i1 t,i2 t, o t) $ H |-
                     if select1 t then P#((i1 ladd i2) t) else Q
```

Then we only have to "run" our `ADD` component at times when `select1 t`, or rather when whatever variable it unifies with when the introduction rule is used, is `true`. There are however a whole family of such rules catering for each conceivable nesting of `if .. then .. else` statements, each of which requires to be proved. The approach to this is to prove a base rule of a particular form and from this base rule to sequentially prove the rules required for successive nesting depths. An ML function

```
mkCond : rule->int->rule
```

has been developed which will generate introduction rules for any given nesting depth when supplied with the base rule for the component and the required depth. Whilst this is not totally general - a completely general function would require not just an integer nesting depth but, say, a sequence of booleans to steer to either the if or the else part of each successive nesting - it served its purpose in our original small example of code synthesis.

The base rule for the `ADD` component is

```
G // H |- P#(o t)
------------------------------------------------
G // ADD#(i1 t,i2 t,o t) $ H |- P#((i1 ladd i2) t)
```

Whilst this works well for simple components, the technique falls down for components such as the integrator. In this case a nested introduction of the component is not possible; the component must be run on every cycle. To see this, suppose select `1't` is `true` then if `t <> 0`, for the component to function correctly at time `1't` we must have

```
store 1't == intgr x gain init t
```

which implies that the component was run on the previous iteration which in turn assumes it was run on the iteration previous to that ....

If an integrator (or any component with introduced state) is not to be run on all cycles then this must be reflected in its L2 functional specification. Adding the boolean signal active to the function and changing the initialisation value to a signal gives

```
fun intgr active x gain init 0   = init 0 ++ gain ** x 0
 |  intgr active x gain init 1't =
          if active 1't then
            let val ynm1 = if active t
                           then intgr active x gain init t
                           else init 1't
            in
               ynm1 ++ gain ** x 1't
            end
          else
               intgr active x gain init t;
```

Here provision is made to re-initialise the integrator value whenever a state transition occurs from inactive to active. The function, being total, also specifies the output from the integrator when `active` is `false`. Here it holds to the output value which obtained the last time active was true. Figure 2 shows two possible implementations. Both introduce another state variable `active_last` which keeps track of the last cycle's value of `active`. This is used to determine when a transition from inactive to active has occurred. The first of these implementations is suitable for replacing an integrator for all time and respects the requirement that the integrator hold its value when `active` is `false`. The second implementation is suitable for a limited form of nested introduction rule applicable only when the integrator is active. In this case the value output from the component when the integrator is inactive is irrelevant and could equally well be specified in the L2 component as `any x:int.TRUE`.

Proving introduction rules and verifying that code conforms to its stated precondition and postcondition requires certain skills. Once a library of proven software components is available however, the process of actually generating code is quite straightforward (with the possible exception of proving the preconditions) as will be explained in the Section 2.4. Thus this approach serves to separate concerns between the formal methods analysts and the control systems analysts.

```
type time = natural;
type ''a signal = time -> ''a;

fun unchanged signal init 0   = signal 0     == init
  | unchanged signal init 1't = signal 1't == signal t;

val INTGR1#(input:int signal,gain:int,store:int signal,output:int signal,
            active:bool signal,active_last:bool signal,init:int signal) =
forall t:time. (if active t then
                   (active_last t == false ->> store t == init t) /\
                   INTGR#(input t,gain,store,output t, t)
                else
                   unchanged output (init 0) t) /\
                   active_last 1't == active t;

val INTGR2#(input:int signal,gain:int,store:int signal,output:int signal,
            active:bool signal,active_last:bool signal,init:int signal) =
forall t:time. (active t == true ->>
                  (active_last t == false ->> store t == init t) /\
                  INTGR#(input t,gain,store,output t, t)) /\
                  active_last 1't == active t;
```

Fig. 2. Two Integrator Implementations

2.4 Code Synthesis

Our method of specification differs from other model based systems (eg. Z, VDM, AMN) in that it does not use a predicate-transformer style but specifies the behaviour for all time. This approach is suited to the loopless, cyclic nature of our software. In effect, the specification is a loop invariant.

To specify, a library of L2 functions which have corresponding software components is required. Each of the functions must be implementable; they respect the limited word length integer arithmetic available, and if they involve the introduction of state variables then they must make suitable initialisations when transitions from inactive to active occur. Whilst the first proviso may be regarded as a nuisance, the second can be a positive benefit in ensuring that such initialisations are not forgotten; each component takes care of itself, helping to ensure that state transitions are seamless. More complex functions are constructed from these components.

The result of applying component introduction to a simple PID[5] controller, the subject of the initial HOLD II[6] study, is shown in Figure 3. The conclusion of the rule gives the initial starting point. It merely specifies a single output as a function of time. Above the line, the first premiss is the result of expanding the

[5] PID stands for Proportional Integrator and Differentiator

[6] *Higher Order Language Demonstrator*, contract ref. FSIA/420, supported by the UK Ministry of Defence.

```
2: G // H |- active 0 == false
1: G // H
 |- forall t.
        (if active t
         then
           (if active_last t
            then
              SCALE#(fmvpe t,dfmvgi t,+1250,o13 t)
              /\ SCALE#(fmvpv t,dfmvgf t,+500,o12 t)
              /\ ADD#(o12 t,dfm2mn t,o11 t)
              /\ ADD#(o12 t,dfm2mx t,o10 t)
              /\ INTGR#(o13 t,store1,+2048,o10 t,o11 t,o1 t,t)
            else
              store t == difinit t /\ (o1 t == upper (intinit t)
           /\ store1 (1't) == intinit t))
           /\ INLINE_DIV#(fmvpe t,+25,o9 t)
           /\ DIFR#(o9 t,store,+25,o t,t)
         else
           unchanged o1 (upper (intinit 0)) t /\ unchanged o +0 t)
       /\ SCALE#(fmvpv t,dfmvgf t,+500,o2 t) /\ SUB#(o1 t,o2 t,o3 t)
       /\ INLINE_DIV#(fmvpe t,+500,o4 t) /\ ADD#(o3 t,o4 t,o5 t)
       /\ ADD#(o5 t,o t,o6 t) /\ SCALE#(o6 t,+2,+1,o7 t)
       /\ LIMIT#(o7 t,cfmcmx t,cfmcmn t,o8 t)
       /\ active_last (1't) == active t /\ output t == o8 t
  ------------------------------------------------------------------------
  G // H
  |- forall t. output t == pid active difinit intinit fmvpe fmvpv
                           dfmvgf dfm2mn dfm2mx dfmvgi cfmcmx cfmcmn t
```

Fig. 3. Simple PID Controller

function definition and introducing the appropriate software component for each of the sub-functions in the specification. The only software components which have been discussed so far are the `ADD` and `INLINE_DIV` components, the latter corresponding to an inlined divide with the simple preconditions here of `25 <> 0` and `500 <> 0`.

Arithmetic is generally protected against overflow in the sense discussed earlier and the `SCALE` component is useful in this respect in multiplying its input by its second parameter to produce a double length result, then dividing by its third parameter to produce a single length result with suitable adjustment for any overflow. The integrator is similar to the one discussed, but has extra parameters and a somewhat more complicated specification to make it practical.

Since they introduce state, the integrator and the differentiator (`DIFR`) both introduce state variables `active` and `active_last` to determine when a state transition occurs, as well as their individual storage locations. The functionality of both of these components becomes distributed throughout the "code". Thus

the two `if` expressions at the start of the component form are attributable jointly to these components (there are no conditional expressions in the specification of function `pid`, and both the integrator and the differentiator have been replaced for all time).

Re-initialisation of the stores occurs when a transition from inactive to active occurs - the update of the the integrator output `o1` here is a consequence of the way the integrator is specified. And, because the integrator and differentiator are active for all time, their outputs are held unchanged when `active` is `false`. The updating of `active_last` with the value of `active` also derives from these two components.

The signals `o`, `o1` to `o13` which connect the software components together are just a consequence of the usage of `o` as an output in the introduction rules. They could be instantiated with more meaningful names, but this would not alter the meaning of the premiss.

The second premiss, that at time 0 the function is inactive, is an artifice to get round the problem of a lack of support for the integers in the LAMBDA rule base. It enabled the introduction rules for the integrator and differentiator to be proved with a dearth of rule support. A useful theory of the integers has subsequently been put in place. This premiss will not be mentioned further.

The associated code which has been automatically generated from the first premiss after reduction to software components is shown in Figure 4. The state variables have been allocated statically. The requirement that the outputs of the differentiator and integrator be unchanging when these components are inactive leads to the static allocation of their outputs and their initialisation at time 0. Since the initial value of the integrator, `o1`, depends on an input variable, it is gated in on the first iteration. In other respects the Ada code matches the software component form fairly closely.

There are, however, significant differences between the Ada code and the software component form. The most obvious is that, whilst the order of the Ada code statements leaves little scope for permutation if the meaning of the program is to be preserved, conjunction is commutative and only the if expressions impose any ordering at all on the component form. The other major difference is that the variables in the component form are integers (and booleans), whereas the Ada variables are signed two's complement with a limited number of bits. One, not completely satisfactory, way of dealing with this semantic gap is to model the generated code in L2. Figure 5 shows an extract of a model of our generated code. The code is thus a function which takes two labelled records one of which contains the inputs, the other the state. The labelled record for the state contains both the statically and dynamically allocated variables, but the latter are undefined on input. The words and longwords in the state are modelled as (isomorphic to) subtypes of the integers satisfying their range constraints. Applying function code with its inputs to the state represents one cycle of execution of procedure `pid`. Functions used in the modelled code consist of the basic software components, and functions which access and update the fields of the labelled records. For example, the latter are of the form

```
with modules ;
package body pid
is

active_last : boolean ;   gate_at_time0 : boolean := true ;
o : short_integer := 0 ;  o1 : short_integer ;
store : short_integer ;   store1 : integer ;

procedure pid (active : boolean ;       difinit : short_integer ;
               intinit : integer ;      fmvpe : short_integer ;
               fmvpv : short_integer ;  dfmvgf : short_integer ;
               dfm2mn : short_integer ; dfm2mx : short_integer ;
               dfmvgi : short_integer ; cfmcmx : short_integer ;
               cfmcmn : short_integer ; output : out short_integer)
is
    o2 : short_integer ; (* declaration of o3 - o12 omitted *)
begin
    if gate_at_time0 then
        o1 := modules.highw(intinit) ; gate_at_time0 := false ;
    end if ;
    if active then
        if active_last then
            o13 := modules.scale (fmvpe, dfmvgi, 1250) ;
            o12 := modules.scale (fmvpv, dfmvgf, 500) ;
            o11 := modules.add (o12, dfm2mn) ;
            o10 := modules.add (o12, dfm2mx) ;
            modules.intgr (o13, store1, 2048, o10, o11, o1) ;
        else
            store := difinit ; o1 := modules.highw(intinit) ;
            store1 := intinit ;
        end if ;
        o9 := fmvpe / 25 ;  modules.difr (o9, store, 25, o) ;
    end if ;

    o2 := modules.scale (fmvpv, dfmvgf, 500); o3 := modules.sub (o1, o2);
    o4 := fmvpe / 500 ;  o5 := modules.add (o3, o4) ;
    o6 := modules.add (o5, o) ;  o7 := modules.scale (o6, 2, 1) ;
    o8 := modules.limit (o7, cfmcmx, cfmcmn) ; active_last := active ;
    output := o8 ;
end pid ;

end pid ;
```

Fig. 4. Code Synthesis Example

```
fun code (input:inrec) (st:state) : state =
let val st =
    if state'gate_at_time0 st then
        let val st = state'o1'up st (highw (inrec'intinit input))
            val st = state'gate_at_time0'up st false
        in
            st
        end
    else [ ... ]
in
    st
end;
```

Fig. 5. Code Modelled in L2

```
state'o12 st  (* Access field o12 of labelled record st:state *)
state'o12'up st value  (* Update field o12 of st with 'value' *)
```

The starting state at each iteration is then modelled. At time 0 all variables in the state are undefined other than those which are explicitly initialised. At times other than 0, the defined variables in the starting state are just those that are statically allocated each of which have the value generated by the previous execution of function code. The state at any time is then given by the function

```
fun st inp t = code (inp t) (start inp t);
```

Having got a model of the code and the state, the proof proceeds by replacing the software components, which are expressed in integers, by even lower level components which work with restricted word lengths. Thus the `ADD` component has the rewrite rule

```
------------------------------------------------------------------
G // is_sw i1 /\ is_sw i2 $ H |-
     ADD#(i1,i2,o) == (o == wordRep (add (wordAbs i1,wordAbs i2))
```

Here `is_sw i1` declares `i1` to be within the signed word range, and `wordRep` and `wordAbs` convert from subtype word to integer and vice versa. The lower level function `add` is given by

```
fun add (input1,input2) =
      wordAbs (wrange (wordRep input1 ++ wordRep input2)) ;
```

The variables in the component form are instantiated with their values in the state, e.g. variable `o` becomes: `fn t =>  wordRep (state'o (st inp t))`.

Following a case analysis on whether or not `t == 0` and a path analysis of the `if` expressions, our original rule is reducible to the form shown in Figure 6. This says that if the output takes its value in the state, the inputs supply the appropriate fields of the input record `inp`, and the inputs are range restricted to

```
val INP#(inp : inrec signal, active : bool signal, difinit : int signal,
         intinit : int signal, fmvpe  : int signal, fmvpv  : int signal,
         dfmvgf : int signal, dfm2mn : int signal, dfm2mx : int signal,
         dfmvgi : int signal, cfmcmx : int signal, cfmcmn : int signal)
 = forall t. inp t ==
     {active = active t, difinit = wordAbs (difinit t),
      intinit = longAbs (intinit t), fmvpe = wordAbs (fmvpe t),
      fmvpv = wordAbs (fmvpv t), dfmvgf = wordAbs (dfmvgf t),
      dfm2mn = wordAbs (dfm2mn t), dfm2mx = wordAbs (dfm2mx t),
      dfmvgi = wordAbs (dfmvgi t), cfmcmx = wordAbs (cfmcmx t),
      cfmcmn = wordAbs (cfmcmn t)} /\
    is_sw (difinit t) /\ is_sl (intinit t) /\ is_sw (fmvpe t) /\
    is_sw (fmvpv t) /\ is_sw (dfmvgf t) /\ is_sw (dfm2mn t) /\
    is_sw (dfm2mx t) /\ is_sw (dfmvgi t) /\ is_sw (cfmcmx t) /\
    is_sw (cfmcmn t) ;

3: G // H |- forall t. output t == wordRep (state'output (st inp t))
2: G // H |- INP#(inp,active,difinit,intinit,fmvpe,fmvpv,dfmvgf,dfm2mn,
                  dfm2mx,dfmvgi,cfmcmx,cfmcmn)
1: G // H |- active 0 == false
   ------------------------------------------------------------------------
   G // H
   |- forall t. output t ==
       pid active difinit intinit fmvpe fmvpv dfmvgf dfm2mn dfm2mx dfmvgi
       cfmcmx cfmcmn t
```

Fig. 6. Verification conditions for the synthesised code

values representable by signed words and signed longwords as appropriate, then the code implements the specification. One should add the proviso: provided the low level software components are implemented by the code.

2.5 Preliminary Experience

The preliminary work detailed in this section was carried out entirely by formal methods specialists. The work addresses a particular class of errors, namely initialisation errors, arithmetic overflow/underflow, type mismatch and the control of activation of components. On the basis of experience we believe that better control of this class of errors has significant cost implications for the Lucas Aerospace development process.

The approach provides very strong guarantees that these errors will not arise in programs that are subjected to test after formal synthesis. This suggests that test effort could either be redirected to providing improved testing of other parts of the program or if that is shown to be unnecessary this could save on development costs.

The approach seems to provide good partitioning between the specialist work involved in building component libraries and proving introduction rules for the

components. This requires specialist work and is time-consuming, however it appears that components are reused in this approach and the cost of verification of the introduction rules can be amortised across a number of projects.

The use of this approach will also change the balance of evidence in the safety case. Formally developed systems will have significantly more deductive evidence of the absence of errors this will lead to a more balanced case.

3 The Experiment

The preliminary case studies have demonstrated that this approach could reap benefits and can be integrated into the existing development process. The next stage in the process of introducing this innovation is a controlled experiment in circumstances that are close to the "real world" situation. To be useful the method must be usable by normal project staff given appropriate training.

Currently we are running a large scale process improvement experiment to assess the utility of the approach. This is funded by the EU ESSI programme. We anticipate the experiment will be complete by the autumn of 1999. This project, known as PCFM, involves collection of metrics from both conventional and formal development processes. The metrics are defined and collected in a controlled manner, so that the two processes can be compared. Our aim is to assess the new process objectively and in a way which de-risks the introduction of new technology.

At the time of writing the project is incomplete. At the moment data is being collected and we are beginning to have enough data to allow comparisons to be made. By the time of the FM99 conference we will have a substantial body of data.

3.1 The Problem

The system being developed is the signal selection and control laws for the *Fuel Metering Unit Smart Electronics Module (FMM SEM)* on the HiPECS project. This project is a demonstrator programme on distributed processing being undertaken in collaboration with the UK Ministry of Defence. The FMM SEM controls the fuel flow and shutoff valve positions to demands received from the core electronics unit. The system is similar to the PID system described in detail earlier but it includes more logic elements that accommodate for errors in sensors and software. This is an interesting extension of the PID case study because it explores the interface between control and logic elements.

3.2 Experimental Setup

The experiment is set up in parallel with a normal project team working to develop a software component. Each project is well instrumented. Data are being collected on where and when faults are uncovered and metrics are being gathered to help compare progress in the two project teams. Data being gathered

falls into two broad categories: Effort data measures developer time through each of the stages of the development process. We believe we will see significant redistribution of effort over the development process (in particular a move away from test effort towards specification and design). The other broad category of data being collected is on product quality. This includes error frequencies and classification. We believe that the formal approach has the potential to greatly reduce the incidence of the errors it is targeting.

3.3 Current Status

At the time of writing the development team have had formal methods training in the use of LAMBDA and have begun to explore the formal specification of the components using LAMBDA to provide simulation facilities and to informally validate the specification against the requirement.

The formal methods experts have constructed a suitable library of formally verified components and introduction rules. These are complete and are currently under review. Once this is complete work on formal code synthesis will commence.

3.4 Preliminary Experience

At this time our experience with the approach in this experiment can only be impressionistic. However we have seen some results that are worth recording:

Difficulties encountered by project staff in forming specifications show we have some usability problems. Despite the similarities, pure functional forms are based on such a different paradigm. Project staff find it hard to adjust to this change. In a small number of cases (e.g. initialisation of historic stores) the imperative model is deeply ingrained. The method being superficially so similar to the conventional approach may paradoxically make it harder for staff to identify and adjust to the differences. One possible line of attack for this problem would be by use of a diagrammatic interface followed by generation of the appropriate L2. DERA Malvern are pursuing some work in this direction [4].

The formal specification gives enhanced visibility of transfer functions. The functional form of the specification means it can be animated to some extent and this has raised some early validation issues.

The code generation method can be expanded to remove some routine calculations (e.g. iteration rates to time constants) which are a potential source of error and/or inconsistency. When available, a secure process should have an easily justifiable business case simply in terms of the amount of reviewing costs one could avoid because informal approach to identifying these inconsistencies is very time consuming.

Being originally targeted at hardware, the LAMBDA environment lacks some features which are essential for this work, but fortunately can be added in (e.g. enhanced rules for manipulation of integers).

References

[1] Adelard. *ASCAD - Adelard Safety Case Development Manual*. Adelard, 1998. ISBN 0 9533771 0 5.

[2] Geoff Barrett. Formal methods applied to a floating-point number systems. *IEEE Transactions on Software Engineering*, 15(5):611–621, May 1989.

[3] B. A. Carre, D. L. Clutterbuck, C. W. Debney, and I. M. O'Neill. SPADE - the thampton Program Analysis and Development Environment. In *Software Engineering Environments*, pages 129–134. Peter Peregrinus, 1986.

[4] P. Caseley, C. O'Halloran, and A. Smith. Explaining code with pictures – a case study. Technical Report DERA/CIS/CIS3/TR990083/1.0 (DRAFT), DERA, 1997.

[5] S. Easterbrook, R Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, 24(1), Jan 1998.

[6] E.M. Mayger and M.P. Fourman. Integration of formal methods with system design. In A. Halaas and P.B. Denyer, editors, *International Conference on Very Large Scale Integration*, pages 59–70, Edinburgh, Scotland, August 1991. IFIP Transactions, North-Holland.

[7] M. P. Fourman. *Formal System Design*, chapter 5, pages 191–236. North-Holland, 1990.

[8] P. Garbett, J. Parkes, M. Shackleton, and S. Anderson. A case study in innovative process improvement: Code synthesis from formal specifications. In *Avionics 98*, 1998.

[9] R. Lutz. Targeting safety-related errors during software requirements analysis. *The Journal of Systems and Software*, 34:223–230, Sept 1996.

[10] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1989.

[11] M.P. Fourman and E.M. Mayger. Formally Based System Design - Interactive hardware scheduling. In G. Musgrave and U. Lauther, editors, *Very Large Scale Integration*, pages 101–112, Munich, Federal Republic of Germany, August 1989. IFIP TC 10/WG10.5 International Conference, North-Holland.

[12] I.M. O'Neill, D.L. Clutterbuck, P.F. Farrow, P.G. Summers, and W.C. Dolman. The formal verification of safety critical assembly code. In *Safety of Computer Control Systems*, pages 115–120. Pergammon Press, 1988.

[13] L. C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828:xvii + 321, 1994.

[14] Requirements and Technical Concepts for Aviation. *Software Considerations in Airborne Systems and Equipment Certification*, Dec 1992. (document RTCA SC167/DO-178B).

[15] S. Finn, M.P. Fourman, and G. Musgrave. Interactive synthesis in HOL-abstract. In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, editors, *International Workshop on Higher Order Logic Theorem Proving and its Applications*, Davis, California, August 1991. IEEE Computer Society, ACM SIGDA, IEEE Computer Society Press.

[16] S. Finn, M.P. Fourman, M.D. Francis, and B. Harris. Formal system design - interactive synthesis based on computer assisted formal reasoning. In Luc J. M. Claesen, editor, *Applied Formal Methods For Correct VLSI Design*, volume 1, pages 97–110. IMEC-IFIP, Elsevier Science Publishers, 1989.

[17] H. Saiedan and L. M. Mc Clanahan. Frameworks for quality software process: SEI capability maturity model. *Software Quality Journal*, 5(1):1, 1996.